Fast Recursive Ensemble Convolution of Haar-like Features

Daniel Wesierski, Maher Mkhinini, Patrick Horain
Département EPH, Telecom SudParis
{first.last}@it-sudparis.eu

Anna Jezierska
Lab. IGM, Université Paris-Est
anna.jezierska@univ-paris-est.fr

Abstract

Haar-like features are ubiquitous in computer vision, e.g. for Viola and Jones face detection or local descriptors such as Speeded-Up-Robust-Features. They are classically computed in one pass over integral image by reading the values at the feature corners. Here we present a new, general, parsing formalism for convolving them more efficiently. Our method is fully automatic and applicable to an arbitrary set of Haar-like features. The parser reduces the number of memory accesses which are the main computational bottleneck during convolution on modern computer architectures. It first splits the features into simpler kernels. Then it aligns and reuses them where applicable forming an ensemble of recursive convolution trees, which can be computed faster. This is illustrated with experiments, which show a significant speed-up over the classic approach.

1. Introduction

This paper addresses the problem of efficiently convolving an image with Haar-like features. The features have become very popular in computer vision during the last decade. They are reminiscent of Haar wavelets and can be thought of as simple, coarse image templates, e.g. edges or bars. When combined, they are able to capture efficiently sparse local image structure, e.g. for face detection [15]. They can serve as smoothing filters and first- and second-order image derivatives which can approximate other kernels accurately [7, 3, 1]. Haar-like features play a very important role in high-level vision, e.g. for pedestrian detection [16] in spatio-temporal domain, for constructing local image descriptors [1, 5], for human limb tracking [11], for side face detection [8] capturing spatial relations between patches, or in pattern matching [10, 13] for image representation in Haar wavelet basis.

The paper is organized as follows. First, previous work is presented in Section 2. Then we formulate our problem in Section 3. In Section 4 we describe an algorithm for parsing Haar-like features into recursive trees of kernels to reduce the number of memory accesses. We achieve this by decomposing the features into smaller kernels and aligning them. An efficient implementation for computing the trees is also proposed. Section 5 experimentally compares the baseline classical approach of Viola and Jones [15] with our approach showing time results on CPU. We conclude in Section 6.

2. Background

In this paper, we adopt the following general definition of Haar-like features. The kernels of Haar-like features are matrices of coefficients obtained after double differentiating piecewise flat patterns.

In 1984, Crow [4] introduced summed-area table (SAT) to the computer graphics community as a generalized method for mip-mapping. In [7], Heckbert used SAT for efficient convolution by repeatedly integrating differentiated box filters in 1D and 2D domains. This was formulated in mathematical terms as $f * k = f^{[n]} * k^{[-n]}$, where $f^{[n]}$ and $k^{[-n]}$ mean n-fold integration and n-fold differentiation of image f and kernel k, respectively.

Unlike [7], where the kernels k were quantized, Simard *et al*. [12] quantized images f forming boxlets. They were then also differentiated to produce trains of impulses along image axes. The coinciding impulses of neighbouring boxlets often cancelled out leading to a reduced representation of the image. Hence, it could be convolved with an arbitrary kernel more rapidly. The algorithm was formulated as $f * k = (f^{[-n]} * k^{[-m]})^{[n+m]}$.

We note this is a general, efficient scheme, which boosts speed performance of convolution primarily by introducing a reduced representation $f^{[-2]}$ of the image f. In their paper, the authors actually do not differentiate the kernel k. This would require recursively integrating the response four times, instead of two, as implied by $f * k = (f^{[-2]} * k^{[-2]})^{[2+2]}$. However, should this be of no concern for any reason, the boxlets scheme could also be applied to the kernel. If the kernel consisted of e.g. an ensemble of Haar-like features, the scheme would create their second-order derivatives $k^{[-2]}$, which we deal with in this paper. Hence, our scheme takes boxlets one step further. Namely, it transforms the boxletization $k^{[-2]}$ of the kernel k to an

ensemble of recursive convolution trees of simpler kernels which eventually require less data to produce exactly the same result as the original kernel $k^{[-2]}$. Therefore, our trees can be well applied together with the boxletization $f^{[-2]}$ of the image f, keeping in mind that the final result would then require recursive integration of the response four times.

Later, Viola and Jones [15] rephrased SAT to integral image and showed how to compute several Haar-like features from it by summing weighted boxes. The attractivity of the integral image comes from the fact that the sum of pixels under e.g. a rectangular area can be computed in constant time at any scale and location by reading the values at four corners of the rectangle. This approach is very simple and thus has become very popular. For example, it has been implemented in the OpenCV library in the framework for rapid object detection [9].

Due to the overall simplicity of this approach, little work has been done on efficient computation of Haar-like features. Since all features are composed of boxes, some approaches consist in first computing boxes and combining them to obtain more complex features [10, 14, 9]. In [10], the authors concentrated on reducing the number of arithmetic operations by introducing a strip sum data structure. However, it needs to be emphasized that nowadays the computation time is bounded primarily by memory accesses [6]. In view of this, we focus on reducing input/output data transfer without regard to the number of required arithmetic operations and we show that this strategy leads to a faster algorithm. Previously, in [14] the authors also considered reducing the number of memory accesses for Haar-like features, though with regard to GPU architecture. While both these methods improve on the naive classic computation [15], they refer only to specific features and are embedded into other algorithms, whereas we give theoretical explanations behind practice and generalize to any Haar-like features.

In this paper we propose a novel algorithm for convolving an image with any set and type of Haar-like features by reducing memory accesses *jointly* for all features. This is challenging when multiple arbitrary features are considered and no assumption is made on their mutual positions. In order to reduce memory accesses during convolution, our idea is to decompose the set of features into smaller kernels, thus forming multi-pass convolutions, and align the kernels within and between passes. This scheme results in an ensemble of recursive convolution trees which reuse previously computed responses of smaller kernels, possibly shared by some subset of features.

3. Problem formulation

In this Section we present our model for reducing inputs and outputs of multiple features during pixel-wise convolution. An input I is a memory read operation, whereas an output O is a memory write operation. We first introduce the parameters and degrees of freedom of the problem. Then, we describe the actions which can be performed on kernels in order to reduce the total sum of inputs and outputs of the convolution.

3.1. Model

Let $\mathcal{K} = \{k_i\}_{i=1}^{N}$ be a set of N convolution kernels of Haar-like features and W be a 2D scanning window with its own coordinate system and origin in the upper left corner, as we employ left-to-right topology on the memory layout. We require all kernels $k_i \in \mathcal{K}$ to be computed explicitly within W. For brevity, we restrict ourselves to case $\text{rank}(k_i) = 1$, but extending it to kernels of higher rank is straightforward. Kernel k_i is a one- or two-dimensional vector or matrix, respectively, parameterized by:

- *input positions*, indicated by non-zero coefficients;
- *size*, determined uniquely by the layout of non-zero coefficients;
- *offset position* $x_i \in \mathbb{N}^2$ from the origin of W, which we write as $k_i(x_i)$.

In the following, apart from $\text{rank}(k_i) = 1$, we have no assumptions concerning the cardinality of $\mathcal{K}$, the number of inputs of k_i, its size nor position in W. We thus say that $\mathcal{K}$ has arbitrary configuration. Consequently, we assign two relation properties to $\mathcal{K}$. A pair of kernels k_i and k_j, where $i, j \leq N$, is:

- *equivalent* if $k_i = \alpha k_j$, where $\alpha \in \mathbb{R}$;
- n-*coinciding* if their n non-zero coefficients are at the same positions in W,

where $n = 0, \ldots, \min(\eta(k_i), \eta(k_j))$ and $\eta(k_i)$ is an operator returning the number of non-zero coefficients of kernel k_i. For instance, window W can contain two equivalent kernels $k_i(x_i)$ and $k_i(x_j)$, which are located at different positions in W, such that $x_i < x_j$. Also, none of their non-zero coefficients may coincide. We then call such kernels 0-coinciding.

Let us now briefly review how to implement a multi-pass convolution scheme, obtained after decomposing a single kernel into P smaller kernels. A small kernel, which is assigned to the first pass, is convolved with the whole image. Then, the second small kernel is convolved with the response obtained after the convolution with the previous kernel. The process is repeated recursively with the remaining $P - 2$ smaller kernels to produce the final response, which is equivalent to the convolution with the original kernel. One can observe this P-pass convolution has in total P outputs, whereas the single-pass convolution has 1 output.

The last output, which writes the final result into memory, is unavoidable for both convolutions. Since we aim at comparing the efficiency of convolution schemes based on the total number of memory accesses required to compute the final result, the count of the final output is ignored in our I/O analysis.

Within the context of memory accesses, which have the cost of several orders of magnitude higher than arithmetic operations on modern CPU architectures, a good strategy for realizing the above multi-pass convolution is to turn it into a buffered recursive single pass. That is, it becomes the single pass convolution over the image, but a multi-pass convolution over the buffer. Namely, in each iteration of the convolution (i.e. at each pixel), a result obtained by one kernel is stored in a temporary variable (ALU register) in order to be used twice in the same iteration between two consecutive passes: 1) it is taken as the last input of the next kernel, and 2) it is output to the buffer so that it can be input to the next kernel in subsequent iteration(s). Therefore, this multi-pass convolution over the buffer reduces the number of inputs of the multi-pass convolution over the image by $P-1$ because it combines the output of the kernel in one pass with the input of the kernel in the next pass in the buffer $P-1$ times. We note that accessing registers has negligible cost.

We formalize the above discussion by introducing three actions which can be performed on a kernel k_i in order to reduce total I/O count of $\mathcal{K}$ while keeping the final results unchanged, namely decomposition, permutation, and alignment. Each action implies cost of c_I inputs and cost of c_O outputs, where $c_I, c_O \in \mathbb{Z}$. The total I/O cost is $c_{I/O} = c_I + c_O$. If $c_{I/O} < 0$, then the actions reduce the total number of inputs and/or outputs w.r.t. the initial configuration of $\mathcal{K}$.

Decomposition of kernel k_i into P_i smaller kernels $k_i = k_i^1 * \ldots * k_i^p \ldots * k_i^{P_i}$ generates a recursive, multi-pass convolution over the buffer, where p indicates the p-th pass (i.e. k_i^1 is convolved with the image as first). The input cost of this action is $c_I = \sum_{p=1}^{P_i} \eta(k_i^p) - (P_i - 1) - \eta(k_i)$, whereas the output cost is $c_O = P_i - 1$. When $P_i > 1$, this action creates 1-coinciding kernels for $p = 1, P_i$ and 2-coinciding kernels for the remaining passes.

Permutation of smaller kernels, e.g. $k_i^1 * \ldots * k_i^{P_i} = k_i^{P_i} * \ldots * k_i^1$, is performed to assign them to specific passes thus changing their positions in W. It has costs $c_I = 0$ and $c_O = 0$. It can lead to reduction of inputs only when combined with alignment with other kernels in the case of multiple features. Yet, if i equivalent kernels ($1 < i \leq N$), unfolded across multiple features after their particular decompositions, are permuted to the same pass and preceded by equivalent kernels in previous passes, then a joint recursion is continued leading to the output cost $c_O = 1 - i$ for this pass.

Alignment of two kernels k_i and k_j is an action which shifts k_i *rightwards* (due to the left-to-right topology) until it coincides with k_j in at least one position. It is analyzed in two possible cases.

1° Single feature. In this case, one can only align last input of kernel k_i^p with the first input of kernel k_i^{p-1} to allow recursion. Aligning a kernel k_i^p with k_i^{p-1} reduces the number of k_i^p inputs by 1. However, this implies that k_i^p is detached from k_i^{p+1}, what increases the number of inputs of k_i^{p+1} by 1. Therefore, this action does not introduce either loss or gain in the number of inputs, and hence it has cost $c_I = 0$. The output cost is $c_O = 0$ as it is not possible to reduce the number of outputs by aligning k_i^p with k_i^{p-1}. There are two special cases. Firstly, shifting k_i^1 has cost $c_I = 1$, as it is not preceded by other kernel. Secondly, since we need to know the value of the feature at specific, predefined location in W, aligning $k_i^{P_i}$ with already shifted $k_i^{P_i-1}$ requires to assign an output to $k_i^{P_i}$ which will be read with single input by additional delta impulse $\delta(x_i)$, placed at the original location of k_i. Hence, if the last kernel is moved, the I/O count increases as $c_I = 1$ and $c_O = 1$. Clearly, shifting a kernel from any pass may reduce total I/O count only in the case of multiple features.

2° Multiple features. This case is more involved for several reasons. Again, we note that it is possible to align kernels of different features only if they are in the same pass and are preceded by equivalent kernels in previous passes to yield recursion. Now, a kernel in a given pass can be *initially* n-coinciding, i.e. before being aligned with another kernel located elsewhere in W. Therefore, if it is aligned with another kernel becoming m-coinciding, the input cost is $c_I = n - m$. This implies that it is possible that aligning kernels may not lead to an improvement, even for equivalent kernels. The output cost for aligning kernels of different features is $c_O = 0$. Moreover, even in the simple case, where all kernels have 2 inputs, aligning them is an NP-complete problem. We motivate our argument by constructing a simple example. Let 4 kernels with 2 inputs have the following sizes: 1, 2, 4, 7. Clearly, there is only one best alignment as $1 + 2 + 4 = 7$, which would result in total of 4 inputs. However, this generally requires to compute all possible sums of integers to decide which subset sum equals to another subset sum. Subset sum problem is NP-complete. So, we try all possible alignments to decide which one yields minimum number of inputs and outputs.

3.2. Example

As a simple example supporting our discussion, consider two kernels of Haar-like features defined as $k_1(1,0) = \begin{bmatrix} +1 & 0 & 0 & 0 & 0 & -1 \\ -3 & 0 & 0 & 0 & 0 & +3 \\ +3 & 0 & 0 & 0 & 0 & -3 \\ -1 & 0 & 0 & 0 & 0 & +1 \end{bmatrix}$ and $k_2(0,5) = \begin{bmatrix} -1 & 0 & 0 & +1 & 0 & +1 & 0 & 0 & -1 \\ +2 & 0 & 0 & -2 & 0 & -2 & 0 & 0 & +2 \\ -1 & 0 & 0 & +1 & 0 & +1 & 0 & 0 & -1 \end{bmatrix}$, which are initially 0-coinciding in $W^{8\times 8}$, as depicted in Fig. 1(a). In classical computation, the number of in-

puts corresponds to the number of non-zero coefficients in the matrices, what here amounts to 20 inputs. After decomposing the features into simpler kernels, we have $k_1 = k_1^x(1,3) * k_1^y(1,1) * k_2^y(1,0) = [+1\ 0\ 0\ 0\ 0\ -1] * [+1\ -2\ +1]^{\mathrm{T}} * [+1\ -1]^{\mathrm{T}}$ and $k_2 = k_1^x(3,7) * k_2^x(0,7) * k_1^y(0,5) = [+1\ 0\ 0\ 0\ 0\ -1] * [-1\ 0\ 0\ +1] * [+1\ -2\ +1]^{\mathrm{T}}$, which amounts to 10 inputs and 4 outputs (Fig. 1(c)). Now, we observe that the features share two equivalent, i.e. one vertical k_1^y, which is explicit (directly indicates side size of two boxes in both features) and one horizontal k_1^x, which is implicit (does not directly indicate side size of any box(es) in feature k_2). By permutation, we assign k_1^x to the first pass, k_1^y to the second pass, and the remaining kernels k_2^x and k_2^y to the third, last pass. Finally, we align the kernels within passes. The kernel $k_1^x(1,3)$ is shifted rightwards to coincide fully with its equivalent kernel $k_1^x(3,7)$ in the first pass. The same is repeated for the next equivalent kernel $k_1^y(1,1)$ in the second pass. Since the remaining kernels are in the last pass and are not equivalent, it is not efficient to align them as it would increase the current total I/O count. Hence, the kernel k_2^y remains at its original position. Concluding, these three passes require 7 inputs and 2 outputs to compute both features exactly (Fig. 1(d)). Theoretically predicted improvement translates into 2.1-fold speed-up.

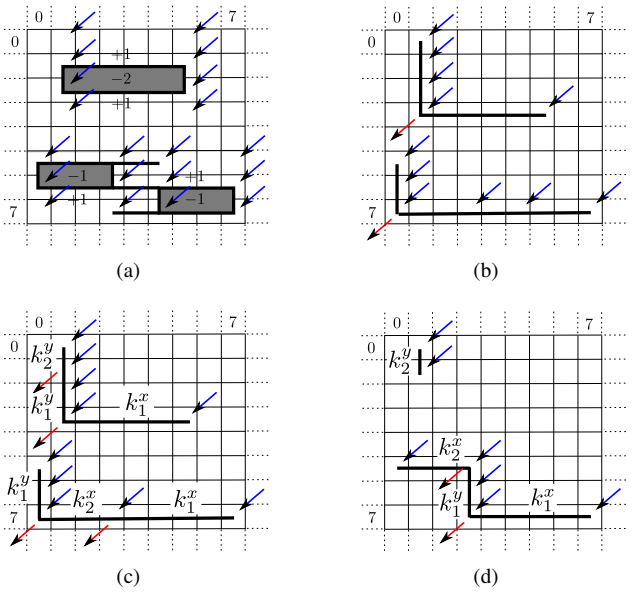

Figure 1. (a,b,c,d) illustrate the steps of decomposition, permutation, and alignment. Each black square indicates non-zero coefficient. The blue arrows incoming to the squares denote inputs from memory, whereas the red outcoming arrows denote outputs to memory. (a) is an example configuration of two features (top - k_1, bottom - k_2), (b) their decomposition from 2D into 1D using SVD, (c) their *further* 1D decomposition which discovers two equivalent kernels, and (d) a recursive convolution tree of kernels after their assignment to and alignment between and within convolution passes.

4. Proposed algorithm

In this Section, we first describe our algorithm. The input of the algorithm is an arbitrary set of Haar-like features assigned to arbitrary positions within a scanning window W. The output is another signal representation of the features in the form of recursive collection of smaller kernels. The algorithm acts like a parser. It splits a particular set of features into smaller kernels, assigns them into passes, aligns them within passes, and creates joint recursions for features if they share equivalent kernels while counting at each step the number of inputs and outputs. We call such a parsed kernel representation an *ensemble of recursive trees*.

In the last part of this Section, we propose a simple yet efficient buffering strategy for implementing the ensemble by increasing the locality of memory reference during convolution.

4.1. Parsing ensembles of recursive trees

We propose an automatic, *off-line* formalism which creates recursive convolution trees of decomposed Haar-like features. They require in total less inputs and outputs to produce the same result as original configuration of the features. First, we procedurally describe our method. The idea is to create recursive multi-pass trees of kernels and align them within each pass in such a way that the total sum of inputs and outputs is minimal. This problem is NP-complete. However, since the number of features is typically not large, it is practical to solve it with a brute-force search. This suggests the following approach:

1. *Decompose* features into smaller kernels in all possible ways such that the number of inputs is reduced.

2. *Assign* kernels of each feature to passes by permuting them.

3. *Align* kernels of all features within and between subsequent passes.

4. *Choose* ensemble of recursive trees with minimal sum of inputs and outputs.

We now describe each step of the procedure in detail.

4.1.1 Decomposing features into smaller kernels

Feature decomposition transforms feature kernel $k_i \in \mathbb{Z}^{X \times Y}$ into a convolution product of vectors $a_{j,t}$, whose first element is equal to 1, last to $j \in \{-1, 1\}$, and the other $(t-2)$ elements are equal to 0, for instance $a_{-1,3} = [+1\ 0\ -1]$. We call vectors $a_{j,t}$ as *primitive kernels* of size t ($t \geq 2, t \in \mathbb{Z}$). P-decomposition of k_i exists if there exist a_{j_p,t_p} s.t. $k_i = \alpha \left(a_{j_1,t_1} * \ldots * a_{j_p,t_p} * (a_{j_{p+1},t_{p+1}})^T * \ldots * (a_{j_P,t_P})^T\right)$,

where $\alpha \in \{-1, 1\}$. The problem is then to find all existing P-decompositions of k_i s.t. $P \leq \eta(k_i)$.

In the following it is assumed that k_i is separable. Thus the SVD $k_i = k_i^x * k_i^y$ exists, where $k_i^x \in \mathbb{Z}^{1\times X}$ and $k_i^y \in \mathbb{Z}^{Y\times 1}$. For simplicity, further presentation concerns decomposition of a vector. We refer to $k_i^{x,y}$ simply as k.

Let $\phi(k, l)$ be an operator returning set $\mathcal{S}$, containing all $a_{j,t}$ such that $t \leq l$ and $\varphi(k, a_{j,t}) = 1$, where

$$\varphi(k_1, k_2) = \begin{cases} 1 & \text{if } \exists\ k' \text{ s.t. } \psi(k_1, k_2) = k' \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

and ψ denotes deconvolution operator i.e. $\psi(k_1, k_2) = k' \Leftrightarrow k_1 = k_2 * k'$. We denote the cardinality of S with L. Note that the P-decomposition of vector k exists if $\max\limits_p(t_p) + \min\limits_p(t_p) > 2\frac{\eta(k)}{P}$, where $\{t_p\}_{p=1}^P$ is a size of p-th primitive kernel forming P-decomposition of k. P-decomposition can yield multiple primitive kernels, e.g. $a_{j_1,t_1} = a_{j_2,t_2}$. Thus we need function $\theta(k, a_{j,t})$ returning $\max\ (m)$ s.t $\varphi(k, (a_{j,t})^m) = 1$, where $(a_{j,t})^m = \underbrace{a_{j,t} * \ldots * a_{j,t}}_{m}$. Finally we introduce function:

$$\xi(\mathcal{S}, \mathcal{M}) = (a_{j_1,t_1})^{m_1} * \ldots * (a_{j_L,t_L})^{m_L}, \tag{2}$$

where $a_{j_l,t_l} \in \mathcal{S}$ and $\mathcal{M} = \{m_l\}_{l=1}^L$ s.t. $m_l = \theta(k, a_{j_l,t_l})$.

We then propose to apply to k the procedure summarized in Algorithm 1, returning $\mathcal{R}_P$ s.t. the convolution product of all P primitive kernels in $\mathcal{R}_P$ gives αk. Using the introduced notation we explain how to create signal s_{full} in each iteration i of Algorithm 1. Firstly, we update $\mathcal{S}^{(i)}$ with $\phi(k^{(i-1)}, \eta(k^{(i-1)}) - p + 1)$. Then we set $\bar{t} = \max(t')$ s.t. $a_{j',t'} \in \mathcal{S}^{(i)}$, and we remove the elements of set $\mathcal{S}^{(i)}$ whose size does not satisfy the condition $t > 2\frac{\eta(k^{(i-1)})}{p} - \bar{t}$. Then the set $\mathcal{M}^{(i)}$ is updated using $m_l^{(i)} = \theta(a_{j_l,k^{(i-1)}}, k^{(i-1)})$. Finally, we calculate s_{full} as $\xi(\mathcal{S}^{(i)}, \mathcal{M}^{(i)})$. One can observe that if $\varphi(s_{full}, k^{(i-1)}) = 0$, then P-decomposition of k does not exist. This simple test limits our search space and allows us to terminate before testing numerous combinations of convolution products of primitive kernels $\in \mathcal{S}$. If the test is successful, we find primitive kernels required to satisfy the condition $\varphi(s'_{full}, k^{(i-1)})$. Since P-decomposition does not exist without these primitive kernels, we know that the resulting set $\mathcal{R}_P$ includes them. If there exists at least one primitive kernel without which the condition is not satisfied, the $k^{(i-1)}$ is simplified and the procedure is repeated iteratively. If not, we are forced to test equality of $k^{(i-1)}$ with all possible combinations of convolution product of p primitive kernels belonging to a set of elements $a_{j_l,k_l} \in \mathcal{S}^{(i)}$ occurring exactly $m_l^{(i)} \in \mathcal{M}^{(i)}$ times. In practice, usually the combinations are tested when L is already small.

Algorithm 1 P-decomposition of k

Set $k^{(0)}$ to k, p to P, $\mathcal{R}_P \longleftarrow \emptyset$
For $i = 1 \ldots$
  Create signal s_{full} knowing $k^{(i-1)}$ and p
  If $\left(\varphi(s_{full}, k^{(i-1)}) = 0\right)$
    P-decomposition of k does not exist; break
  else
    $m = \theta(s_{full}, k^{(i-1)})$
    $s_{temp} \longleftarrow \psi(s_{full}, (k^{(i-1)})^{m-1})$,
    $L \longleftarrow \left|\mathcal{S}^{(i)}\right|$ and $Q \longleftarrow 1$
    For $l = 1 \ldots L$, where $a_{j_l,t_l} \in \mathcal{S}^{(i)}$
      $s'_{full} \longleftarrow \psi(s_{temp}, a_{j_l,t_l})$
      If $\left(\varphi(s'_{full}, k^{(i-1)}) = 0\right)$
        Add a_{j_l,t_l} into resulting set $\mathcal{R}_P$
        Decrease the number of searched primitive kernels $p \longleftarrow p - 1$
        $Q = Q * a_{j_l,t_l}$
    If (Q is equal 1 or p is equal 0)
      Combinatorial update of resulting set $\mathcal{R}_P$
      P-decomposition finished successfully; break
    else
      $k^{(i)} = \psi(k^{(i-1)}, Q)$

The procedure given in Algorithm 1 is repeated for $P = 1, \ldots, \eta(k)$ resulting with the set $\mathcal{D}$ of all possible $\mathcal{R}_P$, i.e. all possible P-decompositions of k, which is used as input to the procedure described in the next Subsection.

4.1.2 Ensembles of trees

This Subsection covers two steps of our parsing procedure, namely assignment of kernels to passes combined with their alignment within and between passes for N Haar-like features. Let $\mathcal{D}_{k_i}$ be the set of all possible P-decompositions of k_i. We augment $\mathcal{D}_{k_i}$ by decompositions into kernels (not necessarily primitive kernels) resulting from all unique convolutions of primitive kernels for each element in $\mathcal{D}_{k_i}$, s.t. their number of inputs $\leq \eta(k_i)$.

The problem now is to choose a single element from each $\mathcal{D}_{k_i}$, where $1 \leq i \leq N$, such that after their particular alignment the number of memory accesses is minimal, thus creating the best ensemble of recursive trees of kernels. We note it is not necessary to permute kernels in each element of $\mathcal{D}_{k_i}$ in all possible ways to enumerate all possible combinations of alignments. This would be straightforward but inefficient. It is possible to align kernels of some subset of N features within given pass only if *all* their kernels in the preceding passes are equivalent, so allowing a recursion. We call this a *proper assignment*. Otherwise, their alignment is not possible and the problem reduces to multiple cases of single features. Of course, all unique kernels

from all features can be aligned within the first pass of the convolution, but the alignments in the next passes depend on the above condition. The pseudo-code is given in Algorithm 2.

Algorithm 2 N-features assignment and alignment

Define ensemble $\mathcal{E}_1 \longleftarrow \emptyset$ *and label it open*
Define set of ensembles $\mathcal{S}_\mathcal{E} \longleftarrow \mathcal{E}_1$
Set merged trees $\left(\mathcal{T}^0_{k_i} \longleftarrow \mathcal{D}_{k_i}\right)_{1 \le i \le N}$
Label each first kernel $\in \mathcal{T}^0_{k_i}$ *with* $\mathcal{E}_1$
For $p = 1 \dots$
  For $i = 1 \dots N$
    For each tree $\mathcal{T}^{p-1}_{k_i}$
      For each branch of tree $\mathcal{T}^p_{k_i}$ from *leaf*
      up to kernel with any $\mathcal{E}$-label in pass p
        Set n *to number of unique kernels in branch*
        Replicate branch $(n-1)$ *times*
        Add new branches to $\mathcal{T}^{p-1}_{k_i}$
        Permute unique kernels of all branches to p
      Merge all branches into single kernel in pass p
          which have equivalent kernels in pass p
  For each $\mathcal{E}_j \in \mathcal{S}_\mathcal{E}$ labeled as open
    For each combination t of $\mathcal{E}_j$-labeled kernels
    of different features in pass p
      Create all possible alignments of kernels
      Set n *to number of alignments*
      Replicate $\mathcal{E}_j$ $(n-1)$ *times. Create set* $\mathcal{A}$ *with* $\mathcal{E}_j$
      Add to each $\mathcal{E}_j \in \mathcal{A}$ *kernels in* t
          with particular alignment
      Update current cost I/O *for each* $\mathcal{E}_j \in \mathcal{A}$
      $\mathcal{S}_\mathcal{E} \longleftarrow \mathcal{S}_\mathcal{E} \cup \mathcal{A}$
  If (Exist $\mathcal{E}_j \in \mathcal{S}_\mathcal{E}$ not having proper assignment)
    For each $\mathcal{E}_j$ not having proper assignment
      For kernels of $\mathcal{E}_j$ *in pass* p *create all*
      combinations of branches from leaf to pass p
      Set n *to number of combinations*
      Label $\mathcal{E}_j$ *as closed*
      Replicate $\mathcal{E}_j$ $(n-1)$ *times. Create set* $\mathcal{A}$ *with* $\mathcal{E}_j$
      Add a combination of branches to each $\mathcal{E}_j \in \mathcal{A}$
      Compute total cost I/O *for each* $\mathcal{E}_j \in \mathcal{A}$
      $\mathcal{S}_\mathcal{E} \longleftarrow \mathcal{S}_\mathcal{E} \cup \mathcal{A}$
  If (Exist $\mathcal{E}_j \in \mathcal{S}_\mathcal{E}$ having proper assignment)
    For each $\mathcal{E}_j$ having proper assignment
      Label children nodes of $\mathcal{E}_j$ *kernels in pass* p *as* $\mathcal{E}_j$
  else
    break

4.1.3 Choosing the best ensemble

After creating all possible unique ensembles of recursive trees $\mathcal{E}_j$, the one is chosen from $\mathcal{S}_\mathcal{E}$ which yields minimal sum of inputs and outputs. However, it is possible that there will be multiple such ensembles. In this special case, we first choose a configuration which has minimal total number of inputs as they are more sparsely referenced than outputs (see Section 4.2 for details). If there are still multiple equal ensembles, we prefer ones with minimal number of inputs in the first pass, then which are more local in this pass, and finally which form buffer of smallest height. If there are any ensembles left after these heuristics, we propose to choose an arbitrary one. We emphasize that these detailed rules are seldom necessary though.

4.2. Implementation: B-channel buffer

The parsed, best ensemble of recursive trees requires multiple outputs, say B, at each iteration of the pixel-wise convolution with the image. Its straightforward implementation would consist of individual circular buffers storing individual outputs, which would be then reused as inputs in subsequent iterations. Buffer sizes would differ from kernel to kernel. For example, a buffer for a vertical kernel would have the height of this kernel and width equal to the image width. On the other hand, a buffer for a horizontal kernel would only have a height of one row and width equal to the width of the kernel. This buffer clearly would occupy less memory than the former. However, since each one would reserve different memory block, such a buffering solution would result in non-local memory reference and thus be cache-unfriendly.

In view of this, we propose to reserve a single contiguous memory block for a buffer which, at each iteration, stores all B outputs of the ensemble of recursive trees in one B-element contiguous data array, similarly to RGB image data structure - hence the appellation B-channel buffer. The inputs defined by kernels parsed into the first pass are read from the image, while the inputs of kernels from the remaining passes are read from the buffer. The input locations are specified by the kernels' positions computed after alignment. Additional predefined offset to particular channel is required for inputs in the buffer. Hence, the width of the buffer equals the image width, while its height depends on a particular alignment of kernels within passes. Indeed, such a buffer occupies more memory than actually required but, in the context of convolution, this is not prohibitive on modern CPU architectures, which suffer from limited memory bandwidth (memory wall) and not from limited memory space. Consequently, such a buffering approach increases locality of memory reference, thus making it a cache-more-friendly-strategy.

5. Results

In this Section, we present the performance of ensembles of recursive trees of kernels parsed by our algorithm. We illustrate their behavior on two examples having practical importance in computer vision. The results are evaluated

in terms of time efficiency by comparing the proposed convolution method with the classical approach of Viola and Jones [15]. We use an integral image of size 4096×4096 in all experiments, though all the results are repeatable for other image sizes. The improvement between both methods is predicted theoretically as:

$$\text{Predicted Improvement} = \frac{I_{class} + 1}{I_{prop} + O_{prop} + 1} \qquad (3)$$

where I_{class} refers to the total number of inputs of the classical method, and I_{prop} and O_{prop} refer to the total number of inputs and outputs of the proposed method, counted per pixel. Classical method requires only inputs to compute all Haar-like features, whereas both methods require an additional output to store the final result. The performence tests were run on 2 Ghz Pentium 4 processor which was connected to 3.5 GB RAM unit through 32-bit data bus with the support of 4 MB cache. The code was compiled with VC++ compiler under Windows environment.

Example SURF. First example illustrates behavior of our algorithm on Haar-like features approximating Hessian of Gaussians in SURF [1]. They are specified by an offset from the origin of the scanning window defined in image coordinate system. Their kernels have the following non-zero coefficients uing standard matrix notation:

1. $k_1(2,0) \in \mathbb{Z}^{10\times 6}$, where: $k_{1,1} = +1$, $k_{1,6} = -1$, $k_{4,1} = -3$, $k_{4,6} = +3$, $k_{7,1} = +3$, $k_{7,6} = -3$, $k_{10,1} = -1$, $k_{10,6} = +1$;

2. $k_2(1,1) \in \mathbb{Z}^{8\times 8}$, where: $k_{1,1} = +1$, $k_{1,4} = -1$, $k_{1,5} = -1$, $k_{1,8} = +1$, $k_{4,1} = -1$, $k_{4,4} = +1$, $k_{4,5} = +1$, $k_{4,8} = -1$, $k_{5,1} = -1$, $k_{5,4} = +1$, $k_{5,5} = +1$, $k_{5,8} = -1$, $k_{8,1} = +1$, $k_{8,4} = -1$, $k_{8,5} = -1$, $k_{8,8} = +1$;

3. $k_3(0,2) \in \mathbb{Z}^{6\times 10}$, where: $k_{1,1} = +1$, $k_{1,4} = -3$, $k_{1,7} = +3$, $k_{1,10} = -1$, $k_{6,1} = -1$, $k_{6,4} = +3$, $k_{6,7} = -3$, $k_{6,10} = +1$.

The proposed approach parses the SURF features jointly, producing recursive trees illustrated in Fig. 2. Their kernels are listed in Table 1, where $0_{[n]}$ denotes a zero vector of size n. One can observe that the number of memory accesses is reduced from 32 to 19, which results in the theoretical time improvement of 1.65. The measured improvement is 1.63, hence confirming the theory.

Example FACE. Similarly, the Haar-like features in the 1st stage of the face detection cascade [9] are defined as:

1. $k_1(1,2) \in \mathbb{Z}^{5\times 19}$, where: $k_{1,1} = -1$, $k_{1,7} = +3$, $k_{1,13} = -3$, $k_{1,19} = +1$, $k_{5,1} = +1$, $k_{5,7} = -3$, $k_{5,13} = +3$, $k_{5,19} = -1$;

2. $k_2(1,7) \in \mathbb{Z}^{10\times 16}$, where: $k_{1,1} = -1$, $k_{1,16} = +1$, $k_{4,1} = +3$, $k_{4,16} = -3$, $k_{7,1} = -3$, $k_{7,16} = +3$, $k_{10,1} = +1$, $k_{10,16} = -1$;

$k_1^x(2,9)$	$[+1\ 0_{[4]}\ -1]$
$k_1^y(2,0)$	$[+1\ 0_{[2]}\ -3\ 0_{[2]}\ +3\ 0_{[2]}\ -1]^{\mathrm{T}}$
$k_2^x(2,9)$	$[+1\ 0_{[2]}\ -1]$
$k_2^y(5,5)$	$[+1\ 0_{[2]}\ -1]^{\mathrm{T}}$
$k_3^x(1,5)$	$[+1\ 0_{[3]}\ -1]$
$k_4^y(1,1)$	$[+1\ 0_{[3]}\ -1]^{\mathrm{T}}$
$k_3^y(6,2)$	$[+1\ 0_{[4]}\ -1]^{\mathrm{T}}$
$k_4^x(0,2)$	$[+1\ 0_{[2]}\ -2\ 0_{[2]}\ +1]$

Table 1. Kernels in recursive trees for SURF example.

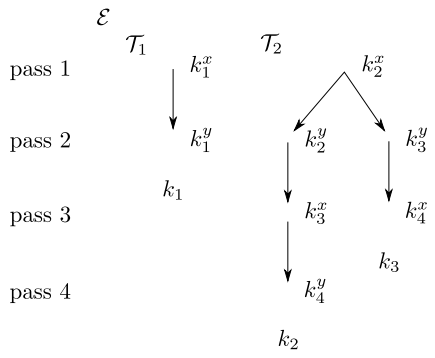

Figure 2. SURF features parsed into ensemble $\mathcal{E}$ of recursive trees. Tree $\mathcal{T}_1$ represents feature k_1, while $\mathcal{T}_2$ represents features k_2 and k_3. The nodes illustrate the decomposed kernels. The directed edges indicate the order of kernels in multi-pass convolution. Each branch of the trees (from root to leaf) corresponds to one feature. For instance, feature k_2 is parsed as: $k_2 = k_2^x * k_2^y * k_3^x * k_4^y$.

3. $k_3(3,7) \in \mathbb{Z}^{5\times 15}$, where: $k_{1,1} = -1$, $k_{1,15} = +1$, $k_{3,1} = +2$, $k_{3,15} = -2$, $k_{5,1} = -1$, $k_{5,15} = +1$,

which are parsed into recursive trees shown in Fig. 3, with kernels given in Table 2. The resulting representation reduces the number of inputs and outputs from 22 to 16, which translates into 1.35-fold speed-up. The measured time improvement is 1.36.

$k_1^y(1,12)$	$[+1\ 0_{[3]}\ -1]^{\mathrm{T}}$
$k_2^x(1,2)$	$[-1\ 0_{[5]}\ +3\ 0_{[5]}\ -3\ 0_{[5]}\ +1]$
$k_1^x(1,16)$	$[+1\ 0_{[14]}\ -1]$
$k_3^y(1,7)$	$[-1\ 0_{[2]}\ +3\ 0_{[2]}\ -3\ 0_{[2]}\ +1]^{\mathrm{T}}$
$k_2^y(1,12)$	$[-1\ 0_{[1]}\ +2\ 0_{[1]}\ -1]^{\mathrm{T}}$
$k_3^x(3,7)$	$[+1\ 0_{[13]}\ -1]$

Table 2. Kernels in recursive trees for FACE example.

The presented experiments are summarized in Table 3. As expected, the measured time improvement is proportional to the ratio between the sum of inputs of the classical approach and the reduced sum of inputs and outputs of our approach. The results clearly indicate that the ensembles of recursive trees, parsed for the above examples using our method, are computed more rapidly than with the classical

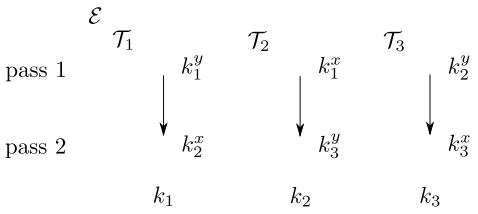

Figure 3. FACE features parsed into trees. Trees $\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3$ represent features k_1, k_2, k_3, respectively.

Example	Approach	I/O	t[ms]	Improvement	
				Predicted	Measured
SURF	Classical	32/0	663.70	1.65	1.63
	Proposed	14/5	407.54		
FACE	Classical	22/0	408.60	1.35	1.36
	Proposed	13/3	300.47		

Table 3. Comparison of classical and our approach in terms of time efficiency for SURF and FACE examples.

approach. The time improvements differ between both examples as each one contains different configuration of Haar-like features. In general, a particular speed-up depends on a particular configuration of the features to be parsed.

6. Conclusions

In this paper, we have presented a strategy for speeding-up convolution with an arbitrary configuration of Haar-like features by parsing them *jointly* into an ensemble of recursive trees of simpler kernels. No approximation of the final result is made. Since the main computational bottleneck of any convolution is memory access, we reduce the number of inputs and outputs jointly for all features. Namely, apart from decomposing the 2D kernels into a horizontal and vertical vector, which is a standard SVD procedure, the parser decomposes them *further* to unfold hidden simpler kernels. This process is controlled under reducing/increasing I/O count criterion in order not to decompose blindly, being neither efficient nor necessary. If the parser discovers cases in which these kernels are shared across the features, it forms a joint recursion tree for them and adds it to the set of other trees as next potential solution. After all potential solutions are formed, the one is chosen which yields minimum total I/O count.

Since we reduce the number of memory accesses jointly for all features inside the scanning window, features of several multiple scales can be parsed jointly as well and convolved with an image in one single pass. This may prove efficient especially for such configurations which share equivalent kernels across features and across scales.

Finally, we emphasize that the recursive trees are not limited to convolutions with integral image representation. This is well justified by integration-differentiation property of convolution published by Heckbert in [7]. They can be convolved also with a differentiated image representation, which could be obtained using e.g. boxlets method developed by Simard *et al*. in [12]. The boxlets scheme would create second-order derivative kernels of Haar-like features, which we take as input to our parsing algorithm. Therefore, our method takes boxlets one step further with respect to multiple Haar-like kernels. Consequently, boxlets method cannot replace our scheme, as it stops at the point where our scheme starts. Whether our method can be applied efficiently also to the boxletized image remains an open problem. We leave it as an interesting future work.

It is possible that future object detectors will require thousands of templates to cope with high variability of object categories [2, 17]. It is therefore desirable to provide tools for computing a set of templates jointly in efficient and rapid manner. We presented a parsing scheme which achieves this goal.

Acknowledgments. This work was partly supported by European FP7 project 216487 CompanionAble and by ERDF project Juliette under CRIF convention 10012367/R.

References

[1] H. Bay, T. Tuytelaars, and L. Van Gool. SURF: Speeded up robust features. In *ECCV*, pages 404–417, May 2006. 1, 7

[2] L. Bourdev and J. Malik. Poselets: Body part detectors trained using 3d human pose annotations. In *ICCV*, 2009. 8

[3] P. Burt. Fast filter transform for image processing. *Computer Graphics and Image Processing*, 16(1):20 – 51, 1981. 1

[4] F. C. Crow. Summed-area tables for texture mapping. *SIGGRAPH Comput. Graph.*, 18:207–212, Jan. 1984. 1

[5] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *CVPR*, pages 886–893, 2005. 1

[6] U. Drepper. What Every Programmer Should Know About Memory. 2007. 2

[7] P. S. Heckbert. Filtering by repeated integration. In *SIGGRAPH*, pages 315–321, New York, NY, USA, 1986. ACM. 1, 8

[8] S. Li, L. Zhu, Z. Zhang, A. Blake, H. Zhang, and H. Shum. Statistical learning of multi-view face detection. In *ECCV*, pages 67–81, 2002. 1

[9] R. Lienhart and J. Maydt. An extended set of Haar-like features for rapid object detection. In *ICIP*, pages 900–903, 2002. 2, 7

[10] W. Ouyang, R. Zhang, and W. Cham. Fast pattern matching using orthogonal haar transform. In *CVPR*, pages 3050–3057, Jun. 2010. 1, 2

[11] D. Ramanan, D. A. Forsyth, and A. Zisserman. Tracking people by learning their appearance. *IEEE Trans. Pattern Anal. Mach. Intell.*, 29:65–81, Jan. 2007. 1

[12] P. Simard, L. Bottou, P. Haffner, and Y. LeCun. Boxlets: A fast convolution algorithm for signal processing and neural networks. In *NIPS*, pages 571–577, 1998. 1, 8

[13] F. Tang, R. Crabb, and H. Tao. Representing images using nonorthogonal Haar-like bases. *IEEE Trans. Pattern Anal. Mach. Intell.*, 29:2120–2134, Dec. 2007. 1

[14] T. Terriberry, L. French, and J. Helmsen. GPU accelerating speeded-up robust features. In *3DPVT*, pages 355–362, Atlanta, GA, USA, Jun. 2008. 2

[15] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *CVPR*, pages 511–518, 2001. 1, 2, 7

[16] P. Viola, M. Jones, and D. Snow. Detecting pedestrians using patterns of motion and appearance. In *ICCV*, pages 734–741, Oct. 2003. 1

[17] Y. Yang and D. Ramanan. Articulated pose estimation using flexible mixtures of parts. In *CVPR*, pages 1385–1392, Jun. 2011. 8